Chapter 1

The nature of programming

1.1 Abstraction

Imagine you wish to explain a play or movie you've seen to a friend. How would you describe it ? There's little point trying to tell them what happens until they know something of the *context*. You might begin by telling them the setting, or *environment*, in which the story unfolds. For example, it might be set in England during the final Napoleonic War. Your friend might then ask who the principal players are, and how their relationship to one another begins. Perhaps there are two young lovers, on the brink of separation. Something of their nature will help, though this mostly emerges through unfolding events. Players have *attributes*, some of which might be important from the outset. Maybe the boy is brave and resolute in his duty to his country, where the girl is equally resolved to serve her ageing parents, who cannot manage without her.

You would surely include only those attributes which matter to the story or its presentation.

The art of story telling has long made use of *objects* of every kind, about which a story can weave. To continue with our example, our young couple might share a ring. Before the boy goes off to war, they break the ring in two so she can recognize her beau on his return. (He may be gone for several years and will have grown to manhood, if he survives.) The ring has great significance to the story and can be whole, broken or matched. Objects too have attributes, subject to change.

Each half of the ring begins in the keeping of the boy or girl, but may not end that way. None may learn of the girl's, but his might be looted, as he lies dying on the battlefield, even by one on his own side. He may simply be robbed, later to recover what was stolen. More than one story is possible.

A new player may arrive, as an old one departs the stage.

©2021 Dr. Ian East

Should her lover return, they may wish to marry. They must then follow an ancient protocol. Each must first seek the blessing of their family, and of both church and community. Wedding banns must be read repeatedly on Sundays, and months pass in which anyone can raise objection.

Here is yet more fabric over which a story may be woven.

Your pal is unlikely to care about details that lack any significance : while the costumes and scenery add much to the performance, they just get in the way of conveying what it was about. Leaving out unnecessary detail is called 'abstraction', and its what most animals do in order to understand, and interact with, the world around them. Different animals need different abstraction ; what is insignificant to one may be a matter of life and death to another. There is also a distinction between levels of abstraction : what doesn't matter for one purpose can be essential for another.

This is precisely what we do when compose a program. We *abstract* behaviour.

The terminology is somewhat different. Instead of a 'player', we refer to a '*process*', though the terms 'object', 'context' and 'attribute' remain unchanged. The script for each process we call a 'procedure', as we would encounter in a training or maintenance manual, or even a cookery book.

Processes own objects, though ownership can change.

What principally distinguishes process and object is that only the former can initiate change or interaction of any kind. An object merely responds to an act of a process. It is primarily passive.

The *state* of an object refers only to that of its attributes, taken together. That of a process is similarly described but must extend to include the point it has reached in its procedure (script). From this we can infer all it has already done and what it is ready now to do.

Just like players coming together to form a play, processes combine to form a *system*. As a play progresses, so does the state of a system, which encompasses that of all its components.

An important issue in abstraction is *concurrency*, which is obvious and fundamental in most systems — so obvious and fundamental that it is easily overlooked. Players in every movie typically act concurrently. It would be odd, and arguably tedious, if they did not. Indeed, it would be difficult in the extreme to describe any workplace without reference to (or assumption of) concurrency.

Concurrency might even be said to be the normal order of everything.

Few programming books have much to say about *communication*, yet it is by far the most powerful way by which to understand and describe any system. When we watch a movie or play, it forms all that is interesting, and perhaps exciting. Players do little else but communicate. They exchange signals, messages and objects. All their interaction can be seen as communication.

We all have a mailbox, and our telephones ring.

Processes run either in sequence (one after the other) or parallel (concurrently). Those in sequence can only interact asynchronously. Perhaps the classic example is an inheritance passed from the deceased to a grandchild as yet unborn, or treasure discovered by the living, buried by a long-dead pirate.

©2021 Dr. Ian East

Communication always follows some sort of *protocol* — a set of governing rules. These are rarely questioned with regard to everyday conversation, but they certainly exist, and not just for courtesy. They enable smooth and efficient interaction. For a minority, they are hard to understand and follow, which demonstrates their reality. In more complicated situations, there will be a more complicated protocol, perhaps requiring a sequence of interactions, as in the steps leading up to marriage. Similarly, before two heads of state may meet, a series of meetings will be required by civil servants or ministers of increasing rank. Both examples make clear that a protocol can break, and communication fail.

From these ideas, we can create a *language* that provides a powerful means of abstraction, which can be used to abstract a very wide, perhaps universal, set of systems efficiently and transparently.

A playwright must compose a script for each player, but the behaviour they describe is rarely that complex. There are more challenging things, like composing a training manual for a combat pilot. Not only is that job more complicated, more sophisticated behaviour is often required. For a start, there will be multiple rôles to perform : flying a challenging plane, engaging an enemy and maintaining a chain of command. The pilot is more like three players in one, each calling for a distinct script. The pilot process can be reduced to subordinate processes, which must somehow *alternate*.

We must also expect to describe *reactive* behaviour. As with any soldier, a pilot may spend long periods waiting, but some events will require a response with minimal delay, or latency. It will be impossible to describe a demanding task like this one without reference also to *prioritization*. Some incoming signals will require interruption of the current task in favour of another, more urgent one. A programming language should therefore offer the means to express *prioritized alternation* between processes according to both state and signals received.

As technology improved, and time scales shrank, pilots could easily become overloaded. The last generation of fighter/bomber typically had a crew of two : one to fly the plane and communicate, the other to monitor the radar and manage weapons. The current generation has a crew of one, who is ably assisted by a computer system, comprising multiple processors, each running multiple processes.

There are prototypes for the next generation flying now, with no human crew at all.

Abstraction comes in degrees, or levels. The machine that executes your program represents the lowest level you will encounter. Because we always target and occasionally employ that level, a following chapter is devoted to it, but we must also be aware of higher levels as well. It will always be easier describing a new system at as high a level as possible, but we must remain precise. Expressing things precisely, yet economically is exactly what mathematics is for, so you should not be surprised at the need for mathematical skills as you broaden your knowledge. For example, logic and set theory are used when discussing exactly what a given process (procedure) does, rather than how it does it.

Deciding precisely what an entire system does is called *specification*.

©2021 Dr. Ian East